Turtlebot3 FastSLAM

Uma Solução Fatorizada para o Problema de Localização e Mapeamento Simultâneo

Alexandre Leal, **Diogo Sampaio**, **Francisco Tavares** e **Marta Valente**
ist1103098, ist1103068, ist1103402 e ist1103574
Sistemas Autónomos
Instituto Superior Técnico
Lisboa, Portugal
alexandre.b.leal@tecnico.ulisboa.pt, diogo.sa.sampaio@tecnico.ulisboa.pt,
francisco.carreira.tavares@tecnico.ulisboa.pt e marta.valente@tecnico.ulisboa.pt

Abstrato — Este trabalho apresenta a implementação e avaliação do algoritmo *FastSLAM*, como solução para o problema do *SLAM*. Em grupo, desenvolvemos e testámos os resultados da nossa implementação do algoritmo em vários ambientes, com o objetivo de confirmar resultados mais fiáveis. Com efeito, começámos por desenvolver um Micro-simulador onde conduzimos os testes em ambiente simulado. Em seguida, dirigimo-nos ao laboratório, onde realizámos os mesmos testes em ambiente real, usando um robô Turtlebot3. Em ambos os casos, analisámos a aplicabilidade do método na construção de mapas suficientemente precisos e na localização do robô em tempo real.

Palavras-chave — FastSLAM, algoritmo, *landmarks*, ROS.

1 Introdução

Este relatório reflete a análise do problema do *SLAM*, os testes efetuados e as soluções propostas.

Assim, o desafio do *SLAM* (*Simultaneous Localization And Mapping*) consiste na criação de um mapa de um ambiente em estudo enquanto se monitoriza a localização de um robô em movimento dentro do mesmo.

O *FastSLAM* é uma das abordagens mais modernas para a resolução desse problema. Com efeito, este usa a filtragem de partículas e oferece vantagens sobre os métodos clássicos baseados em Filtros de *Kalman* Estendidos (*EKF*).

Ao usar uma estratégia inteligente para dividir o problema do *SLAM*, o *FastSLAM* tem uma complexidade linear ou até logarítmica em relação ao número de recursos no ambiente, enquanto os métodos baseados em *EKF* têm uma complexidade exponencial. Além disso, o *FastSLAM* é robusto, conseguindo lidar com associações de dados erradas, uma falha comum que causa divergência nos métodos baseados em *EKF*.

Dito isto, com este projeto, pretendemos desenvolver uma implementação do *FastSLAM* capaz de criar o mapa do ambiente e estimar a posição do robô, utilizando dados provenientes da leitura de uma câmara e da odometria.

2 Métodos e Algoritmos

2.1 FastSLAM

O *FastSLAM* é uma técnica avançada de SLAM que soluciona o problema da localização e mapeamento em ambientes desconhecidos, utilizando uma coleção de partículas para representar a distribuição de probabilidade do estado do sistema.

Cada partícula representa uma possível hipótese da posição do robô e do mapa do ambiente. Para tal, cada partícula mantém um filtro de *Kalman* estendido para cada *landmark* observado, permitindo uma estimativa precisa do mapa. Durante a execução do algoritmo, as partículas são atualizadas, calculando pesos que refletem a probabilidade das observações, dadas as hipóteses de pose e mapa. Esses pesos são usados para reamostrar as partículas, dando mais peso às partículas que melhor correspondem às observações reais. Dessa forma, garante-se que o conjunto de partículas continue a representar de forma precisa a distribuição de probabilidade do estado do sistema ao longo do tempo.

O *FastSLAM* possui, assim, uma vantagem computacional em relação ao *EKF*, sendo que apenas atualiza os filtros para os *landmarks* observados por cada partícula, o que resulta numa redução na complexidade computacional e torna o algoritmo mais adequado para aplicações em tempo real.

O algoritmo 1 apresenta um pseudocódigo que serviu de guia para o desenvolvimento do projeto.

Algoritmo FastSLAM 1.0 com correspondência conhecida

Legenda:

- Posição: x_t
- *Landmark*: c_t
- Observação realizada pelo turtlebot3: z_t
- Ação/Odometria: u_t
- $h(\mu_{t-1}, x_t)$ é a função de observação:
  - › a sua Jacobiana: $h'(\mu_{t-1}, x_t)$
  - › a sua inversa: $h^{-1}(\mu_{t-1}, x_t)$

1: **FastSLAM1.0_correspondência_conhecida**$(z_t, c_t, u_t. X_{t-1})$:
2: **for** k = 1 até N › loop de todas as partículas
3: $\langle x_{t-1}^{[k]}, \langle \mu_{1,t-1}^{[k]}, \Sigma_{1,t-1}^{[k]} \rangle, \dots \rangle$ ⇢ partícula k em X_{t-1}
4: $x_t^{[k]} \sim p(x_t | x_{t-1}^{[k]}, u_t)$ › amostragem da nova partícula
5: $j = c_t$ › leitura da *landmark*
6: **Se** j nunca foi observado antes
7: $\mu_{j,t}^{[k]} = h^{-1}(z_t, x_t^{[k]})$ › cálculo da média de h
8: $H = h'(\mu_{j,t}^{[k]}, x_t^{[k]})$ › cálculo da Jacobiana
9: $\Sigma_{j,t}^{[k]} = (H^{-1})^T Q_t H^{-1}$ › inicialização da covariância
10: $\omega^{[k]} = p_0$ › atribuição de um peso base
11: **Se não**
12: $\hat{z}^{[k]} = h(\mu_{j,t-1}^{[k]}, x_t^{[k]})$ › previsão da posição
13: $H = h'(\mu_{j,t-1}^{[k]}, x_t^{[k]})$ › cálculo da Jacobiana
14: $Q = H\Sigma_{j,t-1}^{[k]} H^T + Q_t$ › covariância da medida
15: $K = \Sigma_{j,t-1}^{[k]} H^T Q^{-1}$ › cálculo do ganho de Kalman
16: $\mu_{j,t}^{[k]} = \mu_{j,t-1}^{[k]} + K(z_t - \hat{z}^{[k]})$ › Atualização da média
17: $\Sigma_{j,t}^{[k]} = (I - KH)\Sigma_{j,t-1}^{[k]}$ › Atualização da covariância
18: $\omega^{[k]} = |2\pi Q|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\left(z_t - \hat{z}^{[k]}\right)^T Q^{-1}\left(z_t - \hat{z}^{[k]}\right)\right\}$
› atribuição do peso à partícula
19: **Terminar o If**
20: $Para\ todos\ j \neq c_t \langle \mu_{j,t}^{[k]}, \Sigma_{j,t}^{[k]} \rangle = \langle \mu_{j,t-1}^{[k]}, \Sigma_{j,t-1}^{[k]} \rangle$
› Dados das *landmarks* não observadas mantém-se inalterados
21: **Terminar o for**
22: **Terminar o for**
23: $X_t = reamostrar\left(\langle x_t^{[k]}, \langle \mu_{1,t}^{[k]}, \Sigma_{1,t}^{[k]} \rangle, \dots, \omega^{[k]} \rangle\right), k = 1, \dots, N$
24: **Retornar** X_t

2.2 Filtro de Partículas

O Filtro de Partículas é utilizado para estimar distribuições de probabilidade em processos não-lineares e não-gaussianos, a partir de amostras. Este método baseia-se na reamostragem por importância, onde é criada uma nova geração de partículas, atribuindo a cada uma um peso de acordo com a sua relevância. Após a atribuição dos pesos, realiza a reamostragem com base nesses valores, permitindo uma aproximação mais precisa da distribuição real do processo modelado.

2.3 Rao-Blackwellization

A *Rao-Blackwellization* é uma técnica usada para reduzir a variância dos estimadores em métodos de Monte Carlo. Ela melhora a eficiência computacional ao decompor uma função de densidade de probabilidade conjunta (PDF) em um produto de funções condicionais, utilizando informações condicionais disponíveis para obter estimadores mais precisos.

No contexto do *FastSLAM*, a *Rao-Blackwellization* é aplicada para decompor a distribuição de probabilidade dos estados do robô e das *landmarks*, e é expressa como:

$$p(s^t, \Theta | n^t, z^t, u^t) = p(s^t | n^t, z^t, u^t) \prod_{n=1}^{N} p(\Theta_n | s^t, n^t, u^t)$$

Aqui, s^t representa a pose do robô no tempo t, Θ são as posições das landmarks, n^t é o conjunto de dados de observações, z^t são as leituras dos sensores, e u^t são as ações passadas do robô.

Esta abordagem resulta em N+1 estimadores recursivos:

1. **Estimador da Pose do Robô:** $p(s^t | n^t, z^t, u^t)$ – que calcula a pose do robô com base nas leituras dos sensores, observações e ações passadas.

2. **Estimadores das *Landmarks*:** $p(\Theta_n | s^t, n^t, u^t)$, para $\mathrm{n} = 1, \dots, \mathrm{N}$ – que determinam as posições das landmarks, condicionadas à estimativa da pose do robô.

Essa estrutura modulariza a gestão e a atualização das estimativas, tornando o processo mais eficiente, tratando as posições das *landmarks* como independentes entre si, sendo condicionadas às posições estimadas do robô.

No *FastSLAM*, a distribuição posterior em cada instante de tempo é representada como um conjunto de partículas ponderadas, cada uma representando uma possível pose do robô. Além disso, cada partícula tem associada a si uma PDF de mapa, que descreve a incerteza nas posições das *landmarks*. Essa PDF é geralmente assumida como aproximadamente Gaussiana e é manipulada utilizando um Filtro de *Kalman* Estendido (EKF).

2.4 Landmarks

As *landmarks* são pontos de referência utilizados na robótica para auxiliar na construção de mapas. Por outras palavras, são pontos previamente selecionados pelos utilizadores, através dos quais o robô pode montar o mapa do ambiente à medida que os reconhece.

Existem dois tipos de *landmarks*: sintéticas e naturais. As primeiras, são pequenos códigos QR formulados em computador, facilmente reconhecidos pelo robô dada a sua aparência binaria. As segundas, são objetos característicos do espaço, por exemplo: a calha das paredes de um corredor (interior) ou bancos de jardim (exterior).

Em ambientes interiores, quando é necessária uma identificação precisa e rápida, é vantajoso utilizar marcadores *ArUco*, devido à facilidade de deteção e decodificação.

Assim, neste projeto, utilizaremos um código em *Python* para gerar e salvar imagens de 10 marcadores *ArUco* diferentes, cada um com um ID único, através de um dicionário predefinido de marcadores. Esses marcadores serão posicionados no ambiente de teste para auxiliar o robô na construção precisa do mapa.

2.5 Modelo de Movimento

A odometria é utilizada para estimar a posição e a orientação de um robô ao longo do tempo, com base em dados de movimento.

O modelo de movimento que vamos utilizar consiste em atualizar a posição (translação) e a orientação (rotação) de um robô a partir de informações de movimento e ruído. Para tal, o modelo utiliza a posição anterior do robô, a informação de odometria (posições estimadas), e a matriz de covariância do ruído de movimento para calcular os novos valores de posição e orientação, tendo em conta as incertezas associadas às medições realizadas.

MODELO DE MOVIMENTO

1: **ModeloMovimento**$(\boldsymbol{u_t}, \boldsymbol{x_{t-1}}, \boldsymbol{R_t})$:
2: #Cálculo das diferenças de translação e rotação:
3: $\delta_{trans} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$
4: $\delta_{rot} = \bar{\theta}' - \bar{\theta}$
5:
6: #Adição de ruído às diferenças calculadas:
7: $\hat{\delta}_{trans} \sim N(\delta_{rot}, \sigma_x^2)$ › $\sigma_x^2 = \sigma_y^2$
8: $\hat{\delta}_{rot} \sim N(\delta_{rot}, \sigma_\theta^2)$
9:
10: #Atualização da pose:
11: $x' = x + \hat{\delta}_{trans} \cos\theta$
12: $y' = y + \hat{\delta}_{trans} \sin\theta$
13: $\theta' = \theta + \hat{\delta}_{rot}$
14:
15: #Retorno da pose nova:
16: **retornar** $x_t = (x', y', \theta')^T$

Logo, o modelo de movimento pode ser representado por esta função:

$$g(x_{t-1}, u_t, R_t) = \begin{bmatrix} x + \hat{\delta}_{trans}\, cos\,\theta \\ y + \hat{\delta}_{trans}\, sin\,\theta \\ \theta + \hat{\delta}_{rot} \end{bmatrix}$$

Onde R_t é a matriz de covariância do ruído $diag(\sigma_x^2, \sigma_x^2, \sigma_x^2)$.

2.6 Modelo de Observação

Para estimar a posição dos *ArUcos* no mundo utilizamos os dados da câmara que o *turtlebot* possui para estimar a distância e o ângulo da *landmark* à câmara do robô. A partir desses dados e sabendo a posição atual do robô no mapa calculamos a posição das *landmarks* observadas no mapa.

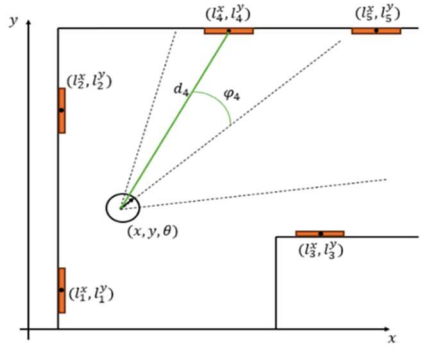

Figura 1:Representação Gráfica do Modelo de Observação.

Com base nisto concluímos que o modelo de observação para o *ArUco* j pode ser representado por esta função:

$$h_t(x_t, d_j, \varphi_j, Q_t) = \begin{bmatrix} l_j^x \\ l_j^y \end{bmatrix} = \begin{bmatrix} x + \hat{d}_j cos\,(\hat{\varphi}_j + \theta) \\ y + \hat{d}_j sen\,(\hat{\varphi}_j + \theta) \end{bmatrix}$$

Onde, $\hat{d}_j \sim N(d_j, \sigma_d^2)$ e $\hat{\varphi}_j \sim N(\varphi_j, \sigma_\varphi^2)$ e Q_t é a matriz de covariância do ruído $diag(\sigma_d^2, \sigma_\varphi^2)$.

A matriz Jacobiana dada pela seguinte função representa como pequenas variações nas medições $\hat{d}_j$ e $\hat{\varphi}_j$ afetam as coordenadas dos *ArUcos* representando muito bem o erro da câmara que será estimada mais à frente em conjunto com o erro da odometria.

$$H_t = \begin{bmatrix} \cos(\hat{\varphi}_j + \theta) & -\hat{d}_j \text{sen}\,(\hat{\varphi}_j + \theta) \\ \text{sen}\,(\hat{\varphi}_j + \theta) & \hat{d}_j \cos(\hat{\varphi}_j + \theta) \end{bmatrix}$$

3 IMPLEMENTAÇÃO

3.1 Estimativa de Incertezas e Parametrização

De forma a conseguirmos avaliar as incertezas na nossa implementação, tivemos de a comparar com dados recolhidos em laboratório.

Em primeiro lugar, testámos a velocidade linear utilizando uma fita métrica para medir a distância real percorrida pelo robô e comparámo-la com a distância calculada pela odometria. De seguida, para a componente da velocidade angular, comparámos, novamente, os resultados da odometria, desta vez com uma rotação de 90º realizada pelo robô. Contudo, não obtivemos resultados muito precisos, devido às dificuldades na medição e na precisão da posição do robô (sendo que não conseguimos garantir uma rotação exata de 90º).

Para avaliar as incertezas na medição da distância e do ângulo, utilizámos outra vez a fita métrica para medir as distâncias entre os marcos e o robô, que comparámos com as distâncias provenientes do código. Para o ângulo, recorremos ainda ao teorema de Pitágoras, onde usámos as distâncias do robô à parede e do chão ao *ArUco* para determinar o valor do ângulo. No entanto, baseamo-nos, mais uma vez, em métodos bastante voláteis, resultando em medições pouco precisas.

Com isto, pudemos então calcular os parâmetros R e Q e usá-los no nosso código.

$$R_t = \begin{pmatrix} 0.204\,m^2 & 0 & 0 \\ 0 & 0.204\,m^2 & 0 \\ 0 & 0 & 0.06854\,rad^2 \end{pmatrix}$$

$$Q_t = \begin{pmatrix} 0.0156\,m^2 & 0 \\ 0 & 0.00762\,rad^2 \end{pmatrix}$$

3.2 Implementação do *FastSLAM*

Para implementar o *FastSLAM*, desenvolvemos as seguintes funções:

1. modelo_movimento():
2. Modelo de movimento das partículas para cada input.
3. prev_particulas(): Previsão da posição das partículas com base no modelo de movimento.
4. obs(): Guarda os dados da observação numa matriz
5. atualiza_com_obs(): Atualização do peso e das posições das *landmarks* das partículas com base nas observações.
6. atualiza_peso(): Cálculo do peso de cada partícula com base nas observações.

7. calc_jacobiana(): Cálculo das matrizes jacobianas
8. adiciona_nova_lm (): Adição de uma nova *landmark*
9. atualiza_FK(): Atualização do filtro de *Kalman* com Cholesky.
10. atualiza_lm(): Cálculo das novas posições das landmarks nas partículas.
11. normaliza_pesos(): Normalização dos pesos das partículas.
12. reamostragem(): Reamostragem das partículas.
13. gaussiana(): Função gaussiana para cálculos estatísticos.

Vídeo completo da nossa implementação:
https://youtu.be/cIP_mYByOS4

3.3 Micro-simulador

Este simulador foi desenvolvido com o objetivo de testar e validar a implementação das funções de predição e atualização do algoritmo *FastSLAM*. Assim, conseguimos simular o movimento de um robô ao longo de um corredor em L, replicando o trajeto que pretendemos fazer no mundo real.

Durante a simulação, o robô deteta e interage com marcadores *ArUco* (posicionados em locais estratégicos) dentro do ambiente simulado, permitindo a correção da trajetória com base nas observações desses marcadores. Este procedimento garante a precisão e a confiabilidade do algoritmo em ambientes controlados antes de ser implementado em situações reais.

3.4 Calibração da câmara

Para calibrarmos a câmara, utilizámos um padrão de xadrez impresso numa folha de papel A4, com o qual capturámos várias imagens a diferentes distâncias e ângulos. Depois, corremos um script de calibração que processa estas imagens e calcula a matriz de calibração e os coeficientes de distorção da câmara. Este processo consiste em detetar os cantos do padrão de xadrez nas imagens capturadas e comparar as distâncias entre estes em todas as imagens, ajustando assim os parâmetros intrínsecos da camara para minimizar os efeitos da distorção ótica observada. Assim, conseguimos assegurar a precisão das medidas e obter um erro mínimo na determinação das distâncias e dos ângulos em relação aos marcadores.

3.5 Rviz e ROS (nós e tópicos)

Para a implementação do algoritmo, precisámos de dados reais que obtivemos através dos tópicos /odom e /raspicam_node/image/compressed provenientes das bags. No tópico /odom, extraímos um vetor de odometria que fornece informações sobre a posição e orientação do robô, enquanto no /raspicam_node/image/compressed, em cada *frame* capturada, verificamos a existência de marcadores *ArUco* presentes na imagem e calculamos a sua distância e direção em relação ao robô. Para tal, desenvolvemos um script que utiliza a biblioteca *OpenCV* para processar as imagens e identificar os marcadores *ArUco*.

No *script*, inicializamos o detetor de marcadores e definimos os parâmetros necessários, como a matriz da câmara e os coeficientes de distorção referidos no ponto 3.4. Assim, durante a sua execução, o script converte as imagens comprimidas em imagens *OpenCV*, identifica os marcadores *ArUco* na imagem e calcula sua posição em relação à câmara.

Uma vez identificados, o script estima a pose de cada marcador, isto é, a sua posição e orientação no espaço tridimensional e, com base nisso, calcula a distância entre a câmara e cada *ArUco*. Além disso, ainda desenha os marcadores na imagem e exibe no ecrã do computador as informações sobre a distância associada a cada marcador.

No entanto, deparámo-nos com um problema de sincronização entre os tópicos /odom e /raspicam_node/image/compressed, que resolvemos assumindo uma velocidade constante entre duas medições de odometria. Assim, utilizámos a velocidade do instante anterior e a diferença de tempo entre a observação e a medida da odometria e aplicámos o modelo de movimento.

4 Resultados Experimentais

4.1 Análise Qualitativa

Realizámos o estudo da nossa implementação no quinto andar da torre Norte. Com efeito, pretendíamos conseguir fazer o mapa completo desse andar, contudo, falhas na rede de internet acabaram por nos limitar apenas a uma secção do piso. Ainda assim, conseguimos obter resultados sólidos que nos permitiram realizar uma boa análise do nosso algoritmo. Dado isto, procurámos balançar estes imprevistos, garantindo resultados consistentes. Para tal, executámos o várias vezes o nosso algoritmo e gravámos também inúmeras *bags*.

Analisando a estimativa da trajetória do robô no mapa, podemos observar na figura 2 a trajetória obtida apenas pela odometria e na figura 3 a trajetória estimada pelo nosso algoritmo do *FastSLAM*.

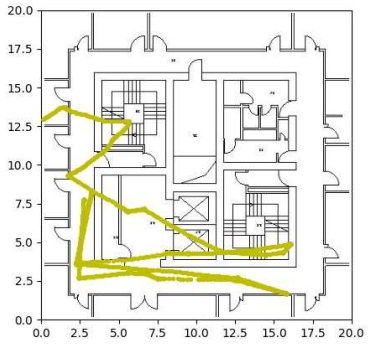

Figura 2: Trajetória do Turtlebot usando dados de Odometria

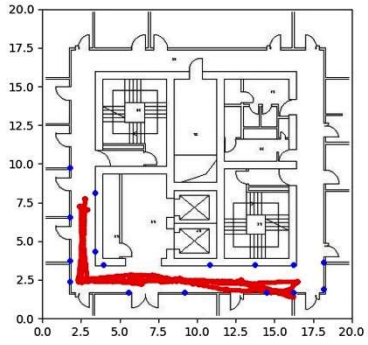

Figura 3: Trajetória do *Turtlebot* usando a nossa implementação do *FastSLAM*

Comparando as duas figuras, podemos concluir que a nossa implementação apresenta uma melhoria na estimativa da trajetória em comparação com a odometria, aproximando-se muito da trajetória real observada. Como não conseguimos completar uma volta completa ao piso, optámos por realizar várias viagens pelo mesmo percurso, com o objetivo de tentar fechar o ciclo, eliminando assim o erro acumulado ao longo do trajeto. Além disso, conduzimos alguns testes no mesmo percurso, mas com velocidades mais baixas, o que resultou numa trajetória, estimada pela odometria, mais precisa.

Analisando agora o mapa criado pelo nosso algoritmo, comparando-o com o mapa criado pelo *gmapping* e ainda com a localização exata de todos os *ArUcos*, podemos verificar, mais uma vez, ótimos resultados.

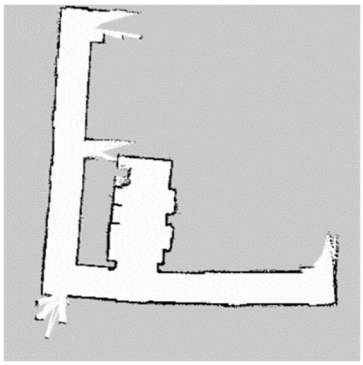

Figura 4: Mapa criado pelo *Gmapping*

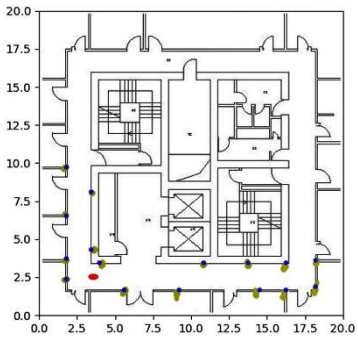

Figura 5: Posição estimada das *landmarks* (amarelo) sobreposta com a posição real das *landmarks* (azul)

4.2 Análise Quantitativa

Para realizar uma boa análise quantitativa, decidimos avaliar a localização e o mapa das *landmarks* em separado.

Para avaliar o mapa das *landmarks* estimadas pelo nosso algoritmo, utilizámos a distância de *Mahalanobis* para calcular o erro entre a distribuição que representa a localização estimada de cada *landmark* e a posição absoluta do *ArUco*. A distância para o *ArUco* i calcula-se a partir da seguinte fórmula:

$$D_i = \sqrt{(p_i - \mu_i)^T \Sigma_i^{-1} (p_i - \mu_i)} =$$

$$= \sqrt{\left(\begin{bmatrix} x_i \\ y_i \end{bmatrix} - \begin{bmatrix} \mu_i^x \\ \mu_i^y \end{bmatrix}\right)^T \Sigma_i^{-1} \left(\begin{bmatrix} x_i \\ y_i \end{bmatrix} - \begin{bmatrix} \mu_i^x \\ \mu_i^y \end{bmatrix}\right)}$$

Onde p_i é o vetor de coordenadas absolutas de cada *ArUco*, μ_i é o vetor com a posição média da distribuição e Σ_i é a matriz de covariância da distribuição. De seguida, podemos observar um gráfico da média da distância de *Mahalanobis* entre cada *landmark* e a posição estimada da mesma representada pela distribuição.

Figura 6: Evolução da média da soma das distâncias entre a posição real e a estimada do *ArUco*

4.3 Problemas e Limitações

A estimativa do erro da velocidade angular não foi exata visto que não tínhamos instrumentos de medição precisos para determinar o valor real. Além disso, nas medições da distância do robô aos *ArUcos*, não considerámos o movimento do robô, o que pode ter introduzido erros significativos nas medições.

De seguida, durante a análise dos dados da odometria no *Excel*, identificámos cerca de quatro valores de velocidade angular instantânea que estavam na ordem das centenas de rad/s, mas apenas para uma medição de cada vez, o que claramente assinalava um erro. Assim, para evitarmos usar essas medidas no nosso algoritmo, optámos por as filtrar, garantindo resultados mais consistentes.

Além disso, como já tínhamos referido no ponto 3.5, os instantes de amostragem da odometria e da câmara não são exatamente iguais, e a nossa implementação do *FastSLAM* requer que estas duas medições ocorram ao mesmo tempo.

Também não obtivemos as posições exatas dos *ArUcos*, tendo recorrido a aproximações a partir do vídeo capturado na gravação das *bags*, o que afetou a nossa análise de resultados.

Por fim, dado que não gravámos o percurso real do *turtlebot*, ficámos restritos à comparação entre a estimativa de localização feita pelo nosso algoritmo e a estimativa apresentada pelo modelo da odometria. Com efeito, gostaríamos de ter esses dados para conseguirmos fazer uma análise mais rigorosa da precisão da nossa solução. Ainda assim, conseguimos notar uma melhoria evidente entre a nossa implementação e o modelo da odometria.

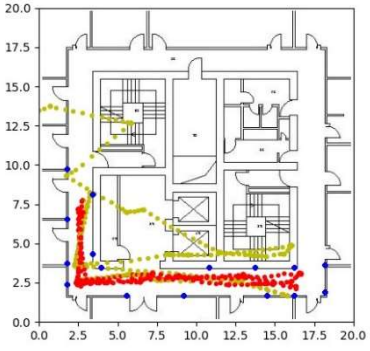

Figura 7: Sobreposição da trajetória estimada pelo nossso algorítmo (vermelho) e pela odometria (amarelo)

5 Conclusão

Neste trabalho, foi desenvolvida uma implementação do algoritmo *FastSLAM* para resolver o problema de *SLAM* (*Simultaneous Localization and Mapping*) utilizando um *Turtlebot* e marcadores *ArUco*. A implementação foi validada através de diversos testes, tanto em simulação como em ambiente real, com resultados promissores.

Os testes demonstraram que a nossa implementação do *FastSLAM* é capaz de criar mapas precisos e de estimar com mais certeza a posição do robô, face à estimativa apenas da odometria. Assim, comparando as trajetórias obtidas pelo nosso algoritmo com as trajetórias reais verificamos uma clara melhoria, aproximando-se muito das trajetórias reais observadas.

Durante o desenvolvimento do projeto, enfrentámos alguns desafios, nomeadamente na calibração da câmara e na sincronização dos dados de odometria e imagem. No entanto, as soluções encontradas permitiram mitigar esses problemas, resultando numa implementação robusta e eficiente.

Em suma, a implementação do *FastSLAM* provou ser eficaz para a navegação e mapeamento em tempo real, apresentando vantagens claras sobre métodos tradicionais baseados em Filtros de *Kalman* Estendidos (*EKF*). Este trabalho conclui a nossa investigação e implementação do algoritmo, evidenciando a sua aplicabilidade prática em robótica autónoma.

6 Referências

[1] R. Ventura, Introduction to the Robot Operating System (ROS). Institute for Systems and Robotics - Lisboa, 2020/2021.

[2] T. Foote, "ROS/Tutorials - ROS Wiki," *wiki.ros.org*, 2022. https://wiki.ros.org/ROS/Tutorials

[3] Thrun et al.: "Probabilistic Robotics", Chapter 13.1-13.3 + 13.8

[4] Montemerlo, Thrun, Kollar, Wegbreit: FastSLAM: *A Factored Solution to the Simultaneous Localization and Mapping Problem*, 2002

[5] Montemerlo and Thrun: *Simultaneous Localization and Mapping with Unknown Data Association Using FastSLAM*, 2003